Databases Project
2023/2024

(2024/03/25: versão 1.1: apenas pacientes podem realizar marcar consultas; na entrega final deve ser submetida também a apresentação a usar na defesa)
(2024/02/25: versão 1.0 do enunciado; pequenos ajustes podem ser realizados nos próximos dias)

Introduction

The goal of this project is to provide students with experience in developing database-centric systems. The project is guided by the industry's best practices for software development; thus, students will experience the main stages of a common software development project, from the very beginning to delivery.

Objectives

After completing this project, students should be able to:

- Understand how a database application development project is organized, planned, and executed
- Master the creation of conceptual and physical data models for supporting application data
- Design, implement, test, and deploy a database system
- Install, configure, manage, and tune a modern relational DBMS
- Understand client and server-side programming in SQL and PL/pgSQL

Groups

The project is to be done in **groups of 3** students.
IMPORTANT: the members of each group **must** be enrolled in PL classes with the same Professor.

Quality attributes for a good project

Your application must make use of:

(a) A transactional relational DBMS (e.g., PostgreSQL)
(b) A distributed database application architecture, providing a REST API
(c) SQL and PL/pgSQL, or similar
(d) Adequate triggers and functions/procedures running on the DBMS side
(e) Good strategies for **managing transactions** and **concurrency conflicts**, and **database security**
(f) Good **error avoidance, detection, and mitigation strategies**
(g) Good documentation

Your application must also respect the functional requirements defined in **Annex A** and execute without "visible problems" or "crashes". To fulfill the objectives of this assignment, you can be as creative as you want, provided you implement the list of required features.

Do not start coding right away - take time to think about the problem and structure your development plan and design.

Plagiarism or any other kind of fraud will not be tolerated

Milestones and deliverables

Midterm defense (20% of the grade) – 23:55 of March 22nd

This defense will take place during the PL classes. No presentation is needed for this defense. It is necessary to sign up for an available time slot for the defense in Inforestudante, before the delivery due date. The list of available slots will be released before the deadline. **All students must be present.** The following

artifacts must be uploaded at Inforestudante until the deadline (one member of the group uploads the artifact but all students must be associated with the submission):

- **Report** with the following information:
  - Name of the project
  - Team members and contacts
  - Brief description of the project
  - Definition of:
    - Transactions (where the concept of transactions is particularly relevant, multiple operations that must succeed or fail as a whole)
    - Potential concurrency conflicts (those that are not directly handled by the DBMS) and the strategy to avoid them
  - Development plan: planned tasks, initial work division per team member, timeline
- **ER diagram**
  - Description of entities, attributes, integrity rules, etc.
- **Relational data model**
  - The physical model of the database (i.e., the tables)
- **The ONDA project, exported as a JSON file.**

Final delivery (80% of the grade) – 23:55 of May 23rd

The project outcomes must be submitted to *Inforestudante* by the deadline. Each group must select a member for performing this task. All submissions must clearly identify the team and the students that compose the group working on the project. Upload the following materials at *Inforestudante*:

- **Final report** with (the following items in a single document):
  - **Installation manual** describing how to deploy and run the software you developed
  - **User manual** describing the submitted Postman collection requests to test the application
  - **Final ER and relational data models**
  - **Development plan:** make sure you specify which tasks were done by each team member and the effort involved (e.g., hours)
  - **All the information, details, and design decisions you consider relevant to understand how the application is built and how it satisfies the requirements of the project**
- **Presentation slides (that will be used during the defense)**
- **Source code and Scripts:**
  - Include the source code, scripts, executable files ,and libraries necessary for compiling and running the software
  - DB creation scripts containing the definitions of tables, constraints, sequences, users, roles, permissions, triggers, functions, and procedures
  - Postman collection with all the requests required to test the application
  - The ONDA project, exported as a JSON file.

Defense – June 3rd to June 7th

- Prepare a 3-minute presentation and 3-minute demo of your software (everything should be ready when entering the defense, otherwise it will be discounted on the available time)
- All students must be present
- Prepare yourself **(individually)** to answer questions regarding all deliverables and implementation details
- Sign up for any available time slot for the defense in *Inforestudante,* before the delivery due date The list of available slots will be released before the deadline for the final delivery

Assessment

- This project accounts for 8 points (out of 20) of the total grade in the Databases course
- Midterm presentation corresponds to 20% of the grade of the project
- Final submission accounts for the remaining 80% of the grade of the project
- The minimum grade is 35%

Technical constraints

- The Entity-relationship model should be created using **ONDA**
- The project is to be developed in **Python**
- The project must use the **PostgreSQL** DBMS
- The interaction with the REST API should be done using **Postman** tool
- The use of ORMs is **not allowed**

Annex A: Hospital Management System – Functional Description

This project aims for you to develop a Hospital Management System (HMS). The development of the database should fit the required functionalities and business restrictions to ensure effective storage and information processing and retrieval.

An HMS is designed to streamline and optimize the operations and workflows within a healthcare facility. At its core, an HMS serves as a centralized platform for managing various aspects of hospital administration, including patient care, scheduling, billing, and resource allocation.

The primary actors of the system include patients and employees, with the latter comprising doctors, nurses, and assistants, each with specific attributes (e.g., a doctor has information related to his/hers medical license; a nurse has an internal hierarchical category). All employees have an employee id and the contract details are also to be stored in the database. The system must be able to manage the patients' appointments and hospitalizations for surgeries (each hospitalization might be associated with multiple surgeries). Each appointment and surgery are conducted by a doctor and can have multiple nurses involved in different specific roles. For each hospitalization there is one nurse that is assigned as responsible. Obviously, no patient, doctor, or nurse can be in two events at the same time. On the other hand, assistants contribute to the operational efficiency by scheduling hospitalizations. Remember that multiple users (e.g., patients) can use the system simultaneously and thus concurrency issues must be considered.

Each appointment and hospitalization can be associated with prescriptions, which define the dosage for each medication (each prescription can comprise multiple medicines). The system also logs a catalog of side effects, and each medicine has multiple side effects with specific occurrences and severity (multiple medicines can have the same side effects, with different probabilities/severity).

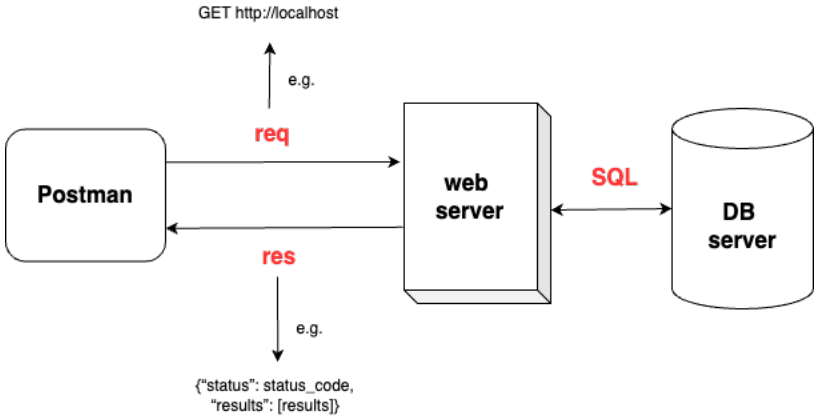
The billing for each appointment and hospitalization must also be stored. For simplicity, let's assume that each appointment and each surgery have a fixed cost. Whenever an appointment is scheduled or a surgery is added to a hospitalization, a bill is created or updated (in the case that a bill had already been created for previous surgery in each hospitalization). This process must be implemented using triggers. Each bill can be split into multiple payments.

Doctors specialize in various medical fields. Specializations can be hierarchically organized (i.e., one specialization may have a parent specialization), allowing for detailed categorization of medical expertise.

Any details that are not specifically defined but that you feel are necessary to develop the database schema should be explicitly defined.

Technical Description

The figure represents a simplified view of the system to be developed. The system must be made available through a REST API that allows the user to access it through HTTP requests (when content is needed, JSON must be used). As it can be seen, the user interacts with the web server through a REST request/response exchange, and in turn, the web server interacts with the database server through an SQL interface (e.g., Psycopg2 in the case of Python). This is one of the most used architectures today and supports many web and mobile applications we use daily. Since the course focuses on the design and operation of a data management system and associated functionalities, the development of web or mobile applications is outside the scope of this work. To use or test the REST API, you must use the REST client Postman (postman.com).



To facilitate the development of the project and focus on the data management system, groups should follow/use the demo code that will be made available. An example of how to test the REST API endpoints using Postman is also provided. In the event of missing details, groups should explicitly identify them in the report.

HTTP works as a request-response protocol. For this work, three main methods might be necessary:

- **GET**: used to request data from a resource
- **POST**: used to send data to create a resource
- **PUT**: used to send data to update a resource

REST API responses should follow a simple but rigid structure, always returning a status code (this is a very simplified version of how a real REST API would work):

- 200 – success
- 400 – error: bad request (request error)
- 500 – error: internal server error (API error)

If there are errors, they must be returned in *errors*, and if there are data to be returned, they must be returned in *results* (as can be seen in the details of the endpoints that follow).

Functionalities to be developed

The *endpoints* of the REST API should be **strictly followed**. Any details that are not defined should be explained in the report.

When the user starts using the platform, he/she can choose between registering a new account or logging in using an existing account. The following *endpoints* should be used.

***NOTICE**: This is a simplified scenario. In a real application, a secure/encrypted connection would be used to send the credentials (e.g., HTTPS).*

Add Patient, Doctor, Nurse, and Assistant. Create a new individual, inserting the data required by the data model. Take into consideration the individual type and attributes to be consider. Do not forget that different types of users have different attributes (e.g., doctors have specialties). Avoid duplicated code. Remember that user details may contain sensitive data.

Endpoint perfeito

Endpoint muito bom - verificar concorrência

Endpoint bom - pode ser melhorado

Endpoint razoável

req **POST** http://localhost:8080/dbproj/register/patient
POST http://localhost:8080/dbproj/register/assistant
POST http://localhost:8080/dbproj/register/nurse
POST http://localhost:8080/dbproj/register/doctor
{"username": username, "email": email, "password": password, (…)}

res {"status": status_code, "errors": errors (if any occurs), "results": user_id (if it succeeds)}

User Authentication. Login using the *username* and the *password* and receive an *authentication token* (e.g., JSON Web Token (JWT), https://jwt.io/introduction) in case of success. This *token* should be included in the *header* of the remaining requests.

req **PUT** http://localhost:8080/dbproj/user
{"username": username, "password": password}

res {"status": status_code, "errors": errors (if any occurs), "results": auth_token (if it succeeds)}

After the authentication, the user can perform the following operations using the obtained *token* during the user authentication (the token should always be passed in every request, either in the body or in authentication headers). **Some operations are restricted to certain types of users.**

Schedule Appointment. Create a new appointment, inserting the data required by the data model. Only a patient can use this endpoint. Remember that for each new appointment a bill should also be created using triggers.

req **POST** http://localhost:8080/dbproj/appointment
"doctor_id": doctor_user_id, "date": date, (…)}

res {"status": status_code, "errors": errors (if any occurs), "results":appointment_id (if it succeeds)}

See Appointments. List all appointments and respective details (e.g., doctor name) of a specific patient Only assistants and the target patient can use this endpoint.

req **GET** http://localhost:8080/dbproj/appointments/{patient_user_id}

res {"status": status_code, "errors": errors (if any occurs), "results": [{"id": appointment1_id, "doctor_id": doctor_user_id, "date": date}, … (if it succeeds)]}

Schedule Surgery. Schedule a new surgery, inserting the data required by the data model. Only assistants can use this endpoint. If *hospitalization_id* is provided, associate with existing hospitalization, otherwise create a new hospitalization. Remember that for each new appointment a bill should also be created (or updated if the hospitalization already exists) using triggers.

req **POST** http://localhost:8080/dbproj/surgery
POST http://localhost:8080/dbproj/surgery/{hospitalization_id}
{"patient_id": patient_user_id, "doctor": doctor_user_id, "nurses": [[nurse_user_id1, role], [nurse_user_id2, role], (…)], "date": date, (…)}

res {"status": status_code, "errors": errors (if any occurs), "results": {"hospitalization_id": hospitalization_id, "surgery_id": surgery_id, "patient_id": patient_user_id, "doctor_id": doctor_user_id, "date": date … (if it succeeds)}}

Get Prescriptions. Get the list of prescriptions and respective details for a patient. Only employees or the targeted patient can use this endpoint.

req **GET** http://localhost:8080/dbproj/prescriptions/{person_id}

res {"status": status_code, "errors": errors (if any occurs), "results": [{"id": presciption_id, "validity": date "posology": ["dose": value, frequency": frequency, "medicine": medicine_name]}, {…}] (if it succeeds)}

Add Prescriptions. When an appointment or hospitalization takes place, a prescription might be necessary. Only doctors can use this endpoint.

Fazer só 1 acesso à BD se possível

req **POST** http://localhost:8080/dbproj/prescription/
{"type": "hospitalization/appointment", "event_id": id, "validity": date, "medicines": [{"medicine": medicine1_name, "posology_dose": value, "posology_frequency": value,

```
      (…)},{“medicine”: medicine2_name, “posology_dose”: value, “posology_frequency”: value,
      (…)}]}
res   {“status”: status_code, “errors”: errors (if any occurs), “results”: prescription_id (…)
      (if it succeeds)}
```

Execute Payment. Pay existing bill. After payment is complete (one bill can have multiple payments) the bill status is updated to “paid”. Only the patient can pay his/her own bills.

```
req   POST http://localhost:8080/dbproj/bills/{bill_id}
      {“amount”: value, “payment_method”: value, (…)}
res   {“status”: status_code, “errors”: errors (if any occurs), “results”: remaining_value}
```

List Top 3 patients. Get the top 3 patients considering the money spent in the Hospital for the current month. The result should discriminate the respective procedures’ details. Just **one** *SQL* query should be used to obtain the information. Only assistants can use this endpoint.

```
req   GET http://localhost:8080/dbproj/top3
res   {“status”: status_code, “errors”: errors (if any occurs), “results”: [
         {“patient_name”: patient1_name, “amount_spent”: value, “procedures”: [{“id”:
      appointment_id, “doctor_id”: doctor_user_id, “date”: date (…)}]},
         {“patient_name”: patient2_name, “amount_spent”: value, “procedures”: [{“id”:
      appointment_id, “doctor_id”: doctor_user_id, “date”: date (…)}]},
      ] (if it succeeds)}
```

Daily Summary. List a count for all hospitalizations details of a given day. Consider, surgeries, payments, and prescriptions. Just **one** *SQL* query should be used to obtain the information. Only assistants can use this endpoint.

```
req   GET http://localhost:8080/dbproj/daily/{year-month-day}
res   {“status”: status_code, “errors”: errors (if any occurs), “results”: {“amount_spent”:
      value, “surgeries”: count, “prescriptions”: count}}
```

Generate a monthly report. Get a list of the doctors with more surgeries each month in the last 12 months. Just **one** *SQL* query should be used to obtain the information. Only assistants can use this endpoint.

```
req   GET http://localhost:8080/dbproj/report
res   {“status”: status_code, “errors”: errors (if any occurs), “results”: [
         {“month”: “month_0”, “doctor”: name, “surgeries”: total_surgeries},
         {“month”: “month_1”, “doctor”: name, “surgeries”: total_surgeries},
         {“month”: “month_2”, “doctor”: name, “surgeries”: total_surgeries},
      (…)
      ]}
```

FINAL REMARKS

- As it is defined in some *endpoints*, the implemented solution should obtain the information using a single SQL *query* on the server side. This does not include potential steps to validate the user authentication. **Nevertheless**, in case you cannot solve it with a single query, it is preferable to use more *queries* than not implementing the *endpoint*.
- The data processing (e.g., order, restrictions) should be done in the *queries* whenever possible (and not in code).
- The transaction control, concurrency, and security will be considered during the evaluation.